Aprendendo python - Unidade 2

Estruturas de dados

ESTRUTURAS DE DADOS EM PYTHON

Em Python, existem objetos que são usados para armazenar mais de um valor, também chamados de estruturas de dados. Cada objeto é capaz de armazenar um tipo de estrutura de dados, particularidade essa que habilita funções diferentes para cada tipo. Portanto, a escolha da estrutura depende do problema a ser resolvido. Para nosso estudo, vamos dividir as estruturas de dados em objetos do tipo sequência, do tipo set, do tipo mapping e do tipo array NumPy. Cada grupo pode possuir mais de um objeto. Vamos estudar esses objetos na seguinte ordem:

- Objetos do tipo sequência: texto, listas e tuplas.
- Objetos do tipo set (conjunto).
- Objetos do tipo mapping (dicionário).
- Objetos do tipo array NumPy.

SEQUÊNCIA DE CARACTERES

Um texto é um objeto da classe str (strings), que é um tipo de sequência. Os objetos da classe str possuem todas as operações apresentadas no Quadro 2.1, mas são objetos imutáveis, razão pela qual não é possível atribuir um novo valor a uma posição específica. Vamos testar algumas operações. Observe o código a seguir.

In [ ]:
```
texto = "Aprendendo Python na disciplina de linguagem de programação."

print(f"Tamanho do texto = {len(texto)}")
print(f"Python in texto = {'Python' in texto}")
print(f"Quantidade de y no texto = {texto.count('y')}")
print(f"As 5 primeiras letras são: {texto[0:6]}")
```

```
Tamanho do texto = 60
Python in texto = True
Quantidade de y no texto = 1
As 5 primeiras letras são: Aprend
```

In [ ]:
```
print(texto.upper())
print(texto.replace("i", 'XX'))
```

```
APRENDENDO PYTHON NA DISCIPLINA DE LINGUAGEM DE PROGRAMAÇÃO.
Aprendendo Python na dXXscXXplXXna de lXXnguagem de programação.
```

In [ ]:
```
print(f"texto = {texto}")
print(f"Tamanho do texto = {len(texto)}\n")

palavras = texto.split()
print(f"palavras = {palavras}")
print(f"Tamanho de palavras = {len(palavras)}")
```

```
texto = Aprendendo Python na disciplina de linguagem de programação.
Tamanho do texto = 60

palavras = ['Aprendendo', 'Python', 'na', 'disciplina', 'de', 'linguagem', 'd
e', 'programação.']
Tamanho de palavras = 8
```

In [ ]:
```
texto = """Operadores de String
Python oferece operadores para processar texto (ou seja, valores de string).
Assim como os números, as strings podem ser comparadas usando operadores de comp
==, !=, <, > e assim por diante.
O operador ==, por exemplo, retorna True se as strings nos dois lados do operado
"""
print(f"Tamanho do texto = {len(texto)}")
texto = texto.lower()
texto = texto.replace(",", "").replace(".", "").replace("(", "").replace(")", ""
lista_palavras = texto.split()
print(f"Tamanho da lista de palavras = {len(lista_palavras)}")

total = lista_palavras.count("string") + lista_palavras.count("strings")

print(f"Quantidade de vezes que string ou strings aparecem = {total}")
```

```
Tamanho do texto = 348
Tamanho da lista de palavras = 58
Quantidade de vezes que string ou strings aparecem = 4
```

LISTAS

Lista é uma estrutura de dados do tipo sequencial que possui como principal característica ser mutável. Ou seja, novos valores podem ser adicionados ou removidos da sequência. Em Python, as listas podem ser construídas de várias maneiras:

- Usando um par de colchetes para denotar uma lista vazia: lista1 = []
- Usando um par de colchetes e elementos separados por vírgulas: lista2 = ['a', 'b', 'c']
- Usando uma "list comprehension": [x for x in iterable]
- Usando o construtor de tipo: list()

In [ ]:
```
vogais = ['a', 'e', 'i', 'o', 'u'] # também poderia ter sido criada usando aspas

for vogal in vogais:
    print (f'Posição = {vogais.index(vogal)}, valor = {vogal}')
```

```
Posição = 0, valor = a
Posição = 1, valor = e
Posição = 2, valor = i
Posição = 3, valor = o
Posição = 4, valor = u
```

In [ ]:
```
# Mesmo resultado.

vogais = []
print(f"Tipo do objeto vogais = {type(vogais)}")

vogais.append('a')
vogais.append('e')
vogais.append('i')
vogais.append('o')
vogais.append('u')
```

```
for p, x in enumerate(vogais):
    print(f"Posição = {p}, valor = {x}")
```

```
Tipo do objeto vogais = <class 'list'>
Posição = 0, valor = a
Posição = 1, valor = e
Posição = 2, valor = i
Posição = 3, valor = o
Posição = 4, valor = u
```

In [ ]:
```
lista = ['Python', 30.61, "Java", 51 , ['a', 'b', 20], "maça"]

print(f"Tamanho da lista = {len(lista)}")

for i, item in enumerate(lista):
    print(f"Posição = {i},\t valor = {item} -----------------> tipo individual =

print("\nExemplos de slices:\n")

print("lista[1] = ", lista[1])
print("lista[0:2] = ", lista[0:2])
print("lista[:2] = ", lista[:2])
print("lista[3:5] = ", lista[3:5])
print("lista[3:6] = ", lista[3:6])
print("lista[3:] = ", lista[3:])
print("lista[-2] = ", lista[-2])
print("lista[-1] = ", lista[-1])
print("lista[4][1] = ", lista[4][1])
```

```
Tamanho da lista = 6
Posição = 0,     valor = Python -----------------> tipo individual = <class 'st
r'>
Posição = 1,     valor = 30.61 -----------------> tipo individual = <class 'flo
at'>
Posição = 2,     valor = Java -----------------> tipo individual = <class 'st
r'>
Posição = 3,     valor = 51 -----------------> tipo individual = <class 'int'>
Posição = 4,     valor = ['a', 'b', 20] -----------------> tipo individual = <c
lass 'list'>
Posição = 5,     valor = maça -----------------> tipo individual = <class 'st
r'>

Exemplos de slices:

lista[1] =  30.61
lista[0:2] =  ['Python', 30.61]
lista[:2] =  ['Python', 30.61]
lista[3:5] =  [51, ['a', 'b', 20]]
lista[3:6] =  [51, ['a', 'b', 20], 'maça']
lista[3:] =  [51, ['a', 'b', 20], 'maça']
lista[-2] =  ['a', 'b', 20]
lista[-1] =  maça
lista[4][1] =  b
```

LIST COMPREHENSION (COMPREENSÕES DE LISTA)

A list comprehension, também chamada de listcomp, é uma forma pythônica de criar uma lista com base em um objeto iterável. Esse tipo de técnica é utilizada quando, dada uma sequência, deseja-se criar uma nova sequência, porém com as informações originais

transformadas ou filtradas por um critério. Para entender essa técnica, vamos supor que tenhamos uma lista de palavras e desejamos padronizá-las para minúsculo. Observe o código a seguir.

In [ ]:
```
linguagens = ["Python", "Java", "JavaScript", "C", "C#", "C++", "Swift", "Go", "
#linguagens = '''Python Java JavaScript C C# C++ Swift Go Kotlin'''.split() # Es

print("Antes da listcomp = ", linguagens)

linguagens = [item.lower() for item in linguagens]

print("\nDepois da listcomp = ", linguagens)
```

```
Antes da listcomp =  ['Python', 'Java', 'JavaScript', 'C', 'C#', 'C++', 'Swif
t', 'Go', 'Kotlin']

Depois da listcomp =  ['python', 'java', 'javascript', 'c', 'c#', 'c++', 'swif
t', 'go', 'kotlin']
```

FUNÇÕES MAP() E FILTER()

Não poderíamos falar de listas sem mencionar duas funções built-in que são usadas por esse tipo de estrutura de dados: map() e filter(). A função map() é utilizada para aplicar uma determinada função em cada item de um objeto iterável. Para que essa transformação seja feita, a função map() exige que sejam passados dois parâmetros: a função e o objeto iterável.

In [ ]:
```
# Exemplo 1
print("Exemplo 1")
linguagens = '''Python Java JavaScript C C# C++ Swift Go Kotlin'''.split()

nova_lista = map(lambda x: x.lower(), linguagens)
print(f"A nova lista é = {nova_lista}\n")

nova_lista = list(nova_lista)
print(f"Agora sim, a nova lista é = {nova_lista}")


# Exemplo 2
print("\n\nExemplo 2")
numeros = [0, 1, 2, 3, 4, 5]

quadrados = list(map(lambda x: x*x, numeros))
print(f"Lista com o número elevado a ele mesmo = {quadrados}\n")
```

```
Exemplo 1
A nova lista é = <map object at 0x000001CF73C0DF10>

Agora sim, a nova lista é = ['python', 'java', 'javascript', 'c', 'c#', 'c++',
'swift', 'go', 'kotlin']


Exemplo 2
Lista com o número elevado a ele mesmo = [0, 1, 4, 9, 16, 25]
```

TUPLAS

As tuplas também são estruturas de dados do grupo de objetos do tipo sequência. A grande diferença entre listas e tuplas é que as primeiras são mutáveis, razão pela qual, com elas, conseguimos fazer atribuições a posições específicas: por exemplo, lista[2] = 'maça'. Por sua vez, nas tuplas isso não é possível, uma vez que são objetos imutáveis.

Em Python, as tuplas podem ser construídas de três maneiras:

- Usando um par de parênteses para denotar uma tupla vazia: tupla1 = ()
- Usando um par de parênteses e elementos separados por vírgulas: tupla2 = ('a', 'b', 'c')
- Usando o construtor de tipo: tuple()

In [ ]:
```
vogais = ('a', 'e', 'i', 'o', 'u')
print(f"Tipo do objeto vogais = {type(vogais)}")

for p, x in enumerate(vogais):
    print(f"Posição = {p}, valor = {x}")
```

```
Tipo do objeto vogais = <class 'tuple'>
Posição = 0, valor = a
Posição = 1, valor = e
Posição = 2, valor = i
Posição = 3, valor = o
Posição = 4, valor = u
```

OBJETOS DO TIPO SET

A tradução"conjunto" para set nos leva diretamente à essência desse tipo de estrutura de dados em Python. Um objeto do tipo set habilita operações matemáticas de conjuntos, tais como: união, intersecção, diferenteça, etc. Esse tipo de estrutura pode ser usado, portanto, em testes de associação e remoção de valores duplicados de uma sequência (PSF, 2020c).

Das operações que já conhecemos sobre sequências, conseguimos usar nessa nova estrutura:

- len(s)
- x in s
- x not in s

In [ ]:
```
vogais_1 = {'aeiou'} # sem uso do construtor
print(type(vogais_1), vogais_1)

vogais_2 = set('aeiouaaaaaa') # construtor com string
print(type(vogais_2), vogais_2)

vogais_3 = set(['a', 'e', 'i', 'o', 'u']) # construtor com lista
print(type(vogais_3), vogais_3)

vogais_4 = set(('a', 'e', 'i', 'o', 'u')) # construtor com tupla
print(type(vogais_4), vogais_4)

print(set('banana'))
```

```
<class 'set'> {'aeiou'}
<class 'set'> {'e', 'u', 'i', 'a', 'o'}
<class 'set'> {'e', 'u', 'i', 'a', 'o'}
<class 'set'> {'e', 'u', 'i', 'a', 'o'}
{'n', 'a', 'b'}
```

In [ ]:
```python
def create_report():
    componentes_verificados = set(['caixas de som', 'cooler', 'dissipador de cal
    componentes_com_defeito = set(['hd', 'monitor', 'placa de som', 'scanner'])

    qtde_componentes_verificados = len(componentes_verificados)
    qtde_componentes_com_defeito = len(componentes_com_defeito)

    componentes_ok = componentes_verificados.difference(componentes_com_defeito)

    print(f"Foram verificados {qtde_componentes_verificados} componentes.\n")
    print(f"{qtde_componentes_com_defeito} componentes apresentaram defeito.\n")

    print("Os componentes que podem ser vendidos são:")
    for item in componentes_ok:
        print(item)


create_report()
```

```
Foram verificados 23 componentes.

4 componentes apresentaram defeito.

Os componentes que podem ser vendidos são:
cpu
placa mãe
joystick
caixas de som
teclado
hub
modem
estabilizador
placa de captura
placa de vídeo
gabinete
memória ram
impressora
dissipador de calor
mouse
microfone
cooler
webcam
no-break
```

OBJETOS DO TIPO MAPPING

As estruturas de dados que possuem um mapeamento entre uma chave e um valor são consideradas objetos do tipo mapping. Em Python, o objeto que possui essa propriedade é o dict (dicionário). Uma vez que esse objeto é mutável, conseguimos atribuir um novo valor a uma chave já existente.

Podemos construir dicionários em Python das seguintes maneiras:

- Usando um par de chaves para denotar um dict vazio: dicionario1 = {}
- Usando um par de elementos na forma, chave : valor separados por vírgulas: dicionario2 = {'one': 1, 'two': 2, 'three': 3}
- Usando o construtor de tipo: dict()

In [ ]:
```
# Exemplo 1 - Criação de dicionário vazio, com atribuição posterior de chave e v
dici_1 = {}
dici_1['nome'] = "João"
dici_1['idade'] = 30

# Exemplo 2 - Criação de dicionário usando um par elementos na forma, chave : va
dici_2 = {'nome': 'João', 'idade': 30}

# Exemplo 3 - Criação de dicionário com uma lista de tuplas. Cada tupla represen
dici_3 = dict([('nome', "João"), ('idade', 30)])

# Exemplo 4 - Criação de dicionário com a função built-in zip() e duas listas, u
dici_4 = dict(zip(['nome', 'idade'], ["João", 30]))


print(dici_1 == dici_2 == dici_3 == dici_4) # Testando se as diferentes construç
```

```
True
```

OBJETOS DO TIPO ARRAY NUMPY

Quando o assunto é estrutura de dados em Python, não podemos deixar de falar dos objetos array numpy. Primeiramente, todas os objetos e funções que usamos até o momento fazem parte do core do interpretador Python, o que quer dizer que tudo está já instalado e pronto para usar. Além desses inúmeros recursos já disponíveis, podemos fazer um certo tipo de instalação e usar objetos e funções que outras pessoas/organizações desenvolveram e disponibilizaram de forma gratuita. Esse é o caso da biblioteca NumPy, criada especificamente para a computação científica com Python.
O NumPy contém, entre outras coisas:

- Um poderoso objeto de matriz (array) N-dimensional.
- Funções sofisticadas.
- Ferramentas para integrar código C/C++ e Fortran.
- Recursos úteis de álgebra linear, transformação de Fourier e números aleatórios.

In [ ]:
```
import numpy

matriz_1_1 = numpy.array([1, 2, 3]) # Cria matriz 1 linha e 1 coluna
matriz_2_2 = numpy.array([[1, 2], [3, 4]]) # Cria matriz 2 linhas e 2 colunas
matriz_3_2 = numpy.array([[1, 2], [3, 4], [5, 6]]) # Cria matriz 3 linhas e 2 co
matriz_2_3 = numpy.array([[1, 2, 3], [4, 5, 6]]) # Cria matriz 2 linhas e 3 colu

print(type(matriz_1_1))

print('\n matriz_1_1 = ', matriz_1_1)
print('\n matriz_2_2 = \n', matriz_2_2)
print('\n matriz_3_2 = \n', matriz_3_2)
print('\n matriz_2_3 = \n', matriz_2_3)
```

```
<class 'numpy.ndarray'>

 matriz_1_1 =  [1 2 3]

 matriz_2_2 =
 [[1 2]
 [3 4]]

 matriz_3_2 =
 [[1 2]
 [3 4]
 [5 6]]

 matriz_2_3 =
 [[1 2 3]
 [4 5 6]]
```

Desafio

Como desenvolvedor em uma empresa de consultoria, você foi alocado para atender um cliente que organiza processos seletivos (concurso público, vestibular, etc.). Essa empresa possui um sistema web no qual os candidatos de um certo processo seletivo fazem a inscrição e acompanham as informações. Um determinado concurso precisou ser adiado, razão pela qual um processo no servidor começou a enviar e-mails para todos os candidatos inscritos. Porém, em virtude da grande quantidade de acessos, o servidor e a API saíram do ar por alguns segundos, o que gerou um rompimento na criação do arquivo JSON com a lista dos candidatos já avisados da alteração.

Por causa do rompimento do arquivo, foram gerados dois novos arquivos, razão pela qual, desde então, não se sabe nem quem nem quantas pessoas já receberam o aviso. Seu trabalho, neste caso, é criar uma função que, com base nas informações que lhe serão fornecidas, retorne uma lista com os e-mails que ainda precisam ser enviados.

Para esse projeto, você e mais um desenvolvedor foram alocados. Enquanto seu colega trabalha na extração dos dados da API, você cria a lógica para gerar a função. Foi combinado, entre vocês, que o programa receberá dois dicionários referentes aos dois arquivos que foram gerados. O dicionário terá a seguinte estrutura: três chaves (nome, email, enviado), cada uma das quais recebe uma lista de informações; ou seja, as chaves nome e email estarão, respectivamente, associadas a uma lista de nomes e uma de emails. por sua vez, a chave enviado estará associada a uma lista de valores booleanos (True-False) que indicará se o e-mail já foi ou não enviado. Veja um exemplo.

```
dict_1 = {
  'nome': ['nome_1'],
  'email': ['email_1'],
  'enviado': [False]
}
```

Você foi alocado para um projeto no qual precisa implementar uma função que, com base em dois dicionários, retorne uma lista com os e-mails que precisam ser enviados aos condidatos de um concurso. Os dicionários serão fornecidos para você no formato combinado previamente. Ao mesmo tempo em que seu colega trabalha na extração dos

dados da API, foi-lhe passado uma amostra dos dados para que você comece a trabalhar na lógica da função. Os dados passados foram:

In [ ]:
```
dados_1 = {
  'nome': ['Sonia Weber', 'Daryl Lowe', 'Vernon Carroll', 'Basil Gilliam', 'Mech
  'email': ['Lorem.ipsum@cursusvestibulumMauris.com', 'auctor@magnis.org', 'at@m
  'enviado': [False, False, False, False, False, False, False, True, False, Fals
}

dados_2 = {
  'nome': ['Travis Shepherd', 'Hoyt Glass', 'Jennifer Aguirre', 'Cassady Ayers',
  'email': ['at@sed.org', 'ac.arcu.Nunc@auctor.edu', 'nunc.Quisque.ornare@nibhAl
  'enviado': [False, False, False, True, True, True, False, True, True, False]
}
```

Mãos ão codigo

In [ ]:
```
def extrair_lista_email(dict_1, dict_2):
    lista_1 = list(zip(dict_1['nome'], dict_1['email'], dict_1['enviado']))
    print(f"Amostra da lista 1 = {lista_1[0]}")

    lista_2 = list(zip(dict_2['nome'], dict_2['email'], dict_2['enviado']))

    dados = lista_1 + lista_2

    print(f"\nAmostra dos dados = \n{dados[:2]}\n\n")

    # Queremos uma lista com o email de quem ainda não recebeu o aviso
    emails = [item[1] for item in dados if not item[2]]

    return emails

emails = extrair_lista_email(dict_1=dados_1, dict_2=dados_2)
print(f"E-mails a serem enviados = \n {emails}")
```

```
Amostra da lista 1 = ('Sonia Weber', 'Lorem.ipsum@cursusvestibulumMauris.com',
False)

Amostra dos dados =
[('Sonia Weber', 'Lorem.ipsum@cursusvestibulumMauris.com', False), ('Daryl Low
e', 'auctor@magnis.org', False)]


E-mails a serem enviados =
 ['Lorem.ipsum@cursusvestibulumMauris.com', 'auctor@magnis.org', 'at@magnaUttin
cidunt.org', 'mauris.sagittis@sem.com', 'nec.euismod.in@mattis.co.uk', 'egestas
@massaMaurisvestibulum.edu', 'semper.auctor.Mauris@Crasdolordolor.edu', 'Donec@
nislMaecenasmalesuada.net', 'Aenean.gravida@atrisus.edu', 'at@sed.org', 'ac.arc
u.Nunc@auctor.edu', 'nunc.Quisque.ornare@nibhAliquam.co.uk', 'dolor.tempus.non@
ipsum.net', 'felis@urnaconvalliserat.org']
```

Algoritimos de busca

Analisando os comandos dos mecanicos de buscas

BUSCA LINEAR (OU BUSCA SEQUENCIAL)

Como o nome sugere, uma busca linear (ou exaustiva) simplesmente percorre os elementos da sequência procurando aquele de destino

In [ ]:
```
import random

def busca_linear(lista, valor):
    for x in lista:
        if x == valor:
            return True
    return False

lista = random.sample(range(1000), 50)
print(sorted(lista))
busca_linear(lista, 50)
```

```
[11, 22, 74, 103, 104, 108, 118, 137, 195, 208, 231, 245, 274, 285, 308, 314, 3
27, 330, 336, 363, 387, 404, 412, 448, 477, 490, 507, 536, 537, 540, 551, 562,
684, 716, 730, 766, 784, 814, 840, 868, 870, 871, 872, 892, 905, 910, 932, 938,
968, 977]
```

Out[ ]: False

In [ ]:
```
def procura_valor(lista, valor):
    tamanho_lista = len(lista)
    for i in range(tamanho_lista):
        if valor == lista[i]:
            return i
    return None

resultado = procura_valor(lista=vogais, valor='a')

if resultado != None:
    print(f"valor encontrado na posicao {resultado}")
else:
    print("valor nao encontrado")
```

```
valor encontrado na posicao 0
```

BUSCA BINARIA

Outro algoritmo usado para buscar um valor em uma sequência é o de busca binária. A primeira grande diferença entre o algoritmo de busca linear e o algoritmo de busca binária é que, com este último, os valores precisam estar ordenados. A lógica é a seguinte:

- Encontra o item no meio da sequência (meio da lista).
- Se o valor procurado for igual ao item do meio, a busca se encerra.
- Se não for, verifica-se se o valor buscado é maior ou menor que o valor central.
- Se for maior, então a busca acontecerá na metade superior da sequência (a inferior é descartada); se não for, a busca acontecerá na metada inferior da sequência (a superior é descartada).

In [ ]:
```
def busca_binaria(lista, valor):
    minimo = 0
    maximo = len(lista)-1
    while minimo <= maximo:
        #encontra o elemento que divide a lista no meio
```

```
        meio = (minimo + maximo) // 2
        #verifica se o valor procurado está a esquerda ou direita do valor centr
        if valor < lista[meio]:
            maximo = meio + 1
        elif valor > lista[meio]:
            minimo = meio + 1
        else:
            return True #valor foi encontrado aqui
    return False # significa que o valor nao foi encontrado

lista = list(range(1, 51))
print(lista)

print('\n',busca_binaria(lista=lista, valor=10))
print('\n',busca_binaria(lista=lista, valor=200))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50]

 True

 False
```

Desafio

Como desenvolvedor em uma empresa de consultoria, você foi alocado para atender um cliente que precisa fazer a ingestão de dados de uma nova fonte e tratá-los. Alocado em uma equipe ágil, você se responsabilizou por implementar uma solução que recebe uma lista de CPFs e a retorna com a transformação de dedup. Além de fazer o dedup, também foi solicitado a você retornar somente a parte numérica do CPF e verificar se eles possuem 11 dígitos. Se não não possuírem, o CPF deve ser eliminado da lista.

In [ ]:
```
#algoritimo de busca_linear
def busca_linear(lista, valor):
    tamanho_lista = len(lista)
    for i in range(tamanho_lista):
        if valor == lista[i]:
            return True
    return False

#função que fará o dedup e os tratamentos no cpf
def criar_lista_dedup(lista):
    lista_dedup = []
    for cpf in lista:
        cpf = str(cpf)
        cpf = cpf.replace("-", "").replace(".", "")
        if len(cpf) == 11:
            if not busca_linear(lista_dedup, cpf):
                lista_dedup.append(cpf)

def testa_funcao(lista_cpfs):
    lista_dedup = criar_lista_dedup(lista_cpfs)
    print(lista_dedup)

lista_cpfs = ['111.111.111-11', '11111111111', '222.222.222-22', '333.333.333-33
testa_funcao(lista_cpfs)
```

```
None
```

Algoritimos de ordenação

APLICAÇÕES DOS ALGORITMOS DE BUSCA

São inúmeras as aplicações que envolvem a necessidade de dados ordenados. Podemos começar mecionando o algoritmo de busca binária, que exige que os dados estejam ordenados. No dia a dia, podemos mencionar a própria lista de chamadas, que possui dados ordenados que facilitam a localização dos nomes. Levando em consideração o hype da área de dados, podemos enfatizar a importância de algoritmos de ordenação, visto que alguns algoritmos de aprendizado de máquina exigem que os dados estejam ordenados para que possam fazer a previsão de um resultado, como a regressão linear.

Em Python, existem duas formas já programadas que nos permitem ordenar uma sequência: a função built-in sorted() e o método sort(), presente nos objetos da classe list. Observe o código a seguir.

In [ ]:
```
lista = [10, 4, 1, 15, -3]

lista_ordenada1 = sorted(lista)
lista_ordenada2 = lista.sort()

print('lista = ', lista, '\n')
print('lista_ordenada1 = ', lista_ordenada1)
print('lista_ordenada2 = ', lista_ordenada2)
print('lista = ', lista)
```

```
lista =  [-3, 1, 4, 10, 15]

lista_ordenada1 =  [-3, 1, 4, 10, 15]
lista_ordenada2 =  None
lista =  [-3, 1, 4, 10, 15]
```

como "programadores usuários", temos essas duas opções para ordenar uma lista em Python, certo? E como profissionais de tecnologia que entendem o algoritmo que está por trás de uma função? Para alcançarmos esse mérito, vamos então conhecer cinco algoritmos de ordenação. A essência dos algoritmos de ordenação consiste em comparar dois valores, verificar qual é menor e colocar na posição correta. O que vai mudar, neste caso, é como e quando a comparação é feita. Para que possamos começar a entender a essência dos algoritmos de ordenação, observemos o código a seguir.

In [ ]:
```
lista = [7, 4]

if lista[0] > lista[1]:
    aux = lista[1]
    lista[1] = lista[0]
    lista[0] = aux

print(lista)
```

```
[4, 7, 5]
```

SELECTION SORT (ORDENAÇÃO POR SELEÇÃO)

O algoritmo selection sort recebe esse nome, porque faz a ordenação sempre escolhendo o menor valor para ocupar uma determinada posição. Para entendermos como funciona o algoritmo, suponha que exista uma fila de pessoas, mas que, por algum motivo, elas precisem ser colocadas por ordem de tamanho, do menor para o maior. Essa ordenação será feita por um supervisor. Segundo o algoritmo selection sort, esse supervisor olhará para cada uma das pessoas na fila e procurará a menor delas. Quando encontrá-la, essa pessoa trocará de posição com a primeira. Veja que, agora, a primeira pessoa da fila está na posição correta, pois é a menor. Em seguida, o supervisor precisa olhar para as demais e escolher a segunda menor pessoa; quando encontrá-la, a pessoa troca de lugar com a segunda. Agora as duas primeiras pessoas da fila estão ordenadas. Esse mecanismo é feito até que todos estejam em suas devidas posições. Portanto, a lógica do algoritmo é a seguinte:

- Iteração 1: percorre toda a lista, procurando o menor valor para ocupar a posição 0.
- Iteração 2: a partir da posição 1, percorre toda a lista, procurando o menor valor para ocupar a posição 1.
- Iteração 3: a partir da posição 2, percorre toda a lista, procurando o menor valor para ocupar a posição 2.
- Esse processo é repetido N-1 vezes, sendo N o tamanho da lista.

In [ ]:
```
def selection_sort(lista):
    n = len(lista)

    for i in range(0, n):
        index_menor = i
        for j in range(i+1, n):
            if lista[j] < lista[index_menor]:
                index_menor = j
        lista[i], lista[index_menor] = lista[index_menor], lista[i]
    return lista
lista = [10, 9, 5, 8, 11, -1, 3]
selection_sort(lista)
```

Out[ ]:
```
[-1, 3, 5, 8, 9, 10, 11]
```

No código da entrada 4, temos uma variável que guarda o tamanho da lista (n). Precisamos de duas estruturas de controle para iterar, tanto para ir atualizando a posição de inserção quanto para achar o menor valor da lista. Usamos a variável i para controlar a posição de inserção e a variável j para iterar sobre os valores da lista, procurando o menor valor. A busca pelo menor valor é feita com o auxílio de uma variável com a qual, quando o menor valor for encontrado, a variável index_menor guardará a posição para a troca dos valores. Quando o valor na posição i já for o menor, então index_menor não se atualiza pelo j. Na linha 9, usamos a atribuição múltipla para fazer a troca dos valores, quando necessário.

Para ajudar na compreensão, veja essa versão do selection sort, na qual criamos uma lista vazia e, dentro de uma estrutura de repetição, usamos a função built-in min() para, a cada iteração, encontrar o menor valor da sequência e adicioná-lo na lista_ordenada. Veja que, a cada iteração, o valor adicionado à nova lista é removido da lista original.

BUBBLE SORT (ORDENAÇÃO POR "BOLHA")

O algoritmo bubble sort (algoritmo da bolha) recebe esse nome porque faz a ordenação sempre a partir do ínicio da lista, comparando um valor com seu vizinho. Para entendermos como funciona o algoritmo, suponha que exista uma fila de pessoas, mas que, por algum motivo, elas precisem ser colocadas por ordem de tamanho, do menor para o maior. Segundo o algoritmo bubble sort, a primeira pessoa da fila (pessoa A), perguntará para o segundo a altura – se o segundo for menor, então eles trocam de lugar. Novamente, a pessoa A perguntará para seu próximo vizinho qual é a altura deste – se esta for menor, eles trocam. Esse processo é repetido até que a pessoa A encontre alguém que é maior, contexto no qual essa nova pessoa vai percorrer a fila até o final fazendo a mesma pergunta. Esse processo é repetido até que todas as pessoas estejam na posição correta. A lógica do algoritmo é a seguinte:

- Iteração 1: seleciona o valor na posição 0 e o compara com seu vizinho – se for menor, há troca; se não for, seleciona o próximo e compara, repentindo o processo.
- Iteração 2: seleciona o valor na posição 0 e compara ele com seu vizinho, se for menor troca, senão seleciona o próximo e compara, repentindo o processo.
- Iteração N - 1: seleciona o valor na posição 0 e o compara com seu vizinho – se for menor, há troca; se não for, seleciona o próximo e compara, repentindo o processo.

A variável i é usada para controlar a repetição de todo processo; e a variável j é usada para fazer as comparações entre os elementos. Na iteração 1, o valor 10 ocupará a posição 0, sendo, então, comparado ao valor da posição 1 (9) – como 10 é maior, então é feita a troca. Em seguida, o valor 10 é comparado a seu novo vizinho (5) – como é maior, é feita a troca. Em seguida, o valor 10 é comparado ao novo vizinho (8) – como é menor, é feita a troca. Em seguida, como não há mais vizinhos, a primeira iteração encerra. A segunda iteração, começa novamente com a comparação do valor das posições 0 e 1 – novamente, o valor 9 (posição 0) é maior que 5 (posição 1), então é feita a troca. Em seguida, 9 é comparado ao próximo vizinho 8 – como é maior, é feita a troca. Em seguida, 9 é comparado ao seu novo vizinho 10, como é menor, não é feita a troca. A terceira iteração começa novamente comparando o valor das posições 0 e 1, porém, agora, 5 é menor que 8, razão pela qual não se faz a troca. Na sequência, o valor 8 é comparado ao seu vizinho 9 – como é menor, não é feita a troca. Na sequência, o valor 9 é comparado ao seu vizinho 10 – como é menor, não é feita a troca, quando, então, se encerra o algoritmo.

In [ ]:
```
def executar_bubble_sort(lista):
    n = len(lista)
    for i in range(n-1):
        for j in range(n-1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista

lista = [10, 9, 5, 8, 11, -1, 3]
executar_bubble_sort(lista)
```

Out[ ]: `[-1, 3, 5, 8, 9, 10, 11]`

INSERTION SORT (ORDENAÇÃO POR INSERÇÃO)

O algoritmo insertion sort (algoritmo de inserção) recebe esse nome porque faz a ordenação pela simulação da inserção de novos valores na lista. Para entendermos como funciona o algoritmo, imagine um jogo de cartas para a execução do qual cada jogador começa com cinco cartas e, a cada rodada, deve pegar e inserir uma nova carta na mão. Para facilitar, o jogador opta por deixar as cartas ordenadas em sua mão, razão pela qual, a cada nova carta que precisar inserir, ele olha a sequência da esquerda para a direita, procurando a posição exata para fazer a inserção. A lógica do algoritmo é a seguinte:

- Início: parte-se do princípio de que a lista possui um único valor e, consequentemente, está ordenada.
- Iteração 1: parte-se do princípio de que um novo valor precisa ser inserido na lista; nesse caso, ele é comparado com o valor já existente para saber se precisa ser feita uma troca de posição.
- Iteração 2: parte-se do princípio de que um novo valor precisa ser inserido na lista; nesse caso, ele é comparado com os valores já existentes para saber se precisam ser feitas trocas de posição.
- Iteração N: parte-se do princípio de que um novo valor precisa ser inserido na lista; nesse caso, ele é comparado com todos os valores já existentes (desde o início) para saber se precisam ser feitas trocas de posição.

variável i é usada para guardar a posição do elemento que queremos inserir, e a variável j é usada para encontrar a posição correta a ser inserida. Ou seja, j será usada para percorrer as posições já preenchidas. O primeiro valor da lista é 10, razão pela qual se parte do princípio de que a lista só possui esse valor e está ordenada.

Na primeira iteração, em que o valor 9 precisa ser inserido na lista, ele deve ser inserido na posição 0 ou 1? Como é menor, é feita a troca com o 10. Na segunda iteração, queremos inserir o valor 5, razão pela qual ele é comparado a todos seus antecessores para encontrar a posição correta – como é menor que todos eles, então 5 é alocado na posição 0 e todos os demais são deslocados "para frente". Na terceira iteração, queremos inserir o valor 8, razão pela qual ele é comparado a todos seus antecessores para encontrar a posição correta e, então, é alocado para a posição 1. Todos os outros valores são deslocados "para frente".

In [ ]:
```python
def executar_insertion_sort(lista):
    n = len(lista)
    for i in range(1, n):
        valor_inserir = lista[i]
        j = i - 1

        while j >= 0 and lista[j] > valor_inserir:
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = valor_inserir

    return lista
```

```
lista = [10, 9, 5, 8, 11, -1, 3]
executar_insertion_sort(lista)
```

Out[ ]: `[-1, 3, 5, 8, 9, 10, 11]`

MERGE SORT (ORDENAÇÃO POR JUNÇÃO)

O algoritmo merge sort recebe esse nome porque faz a ordenação em duas etapa: (i) divide a lista em sublistas; e (ii) junta (merge) as sublistas já ordenadas. Esse algoritmo é conhecido por usar a estratégia "dividir para conquistar" (STEPHENS, 2013). Essa estratégia é usada por algoritmos de estrutura recursiva: para resolver um determinado problema, eles se chamam recursivamente uma ou mais vezes para lidar com subproblemas intimamente relacionados. Esses algoritmos geralmente seguem uma abordagem de dividir e conquistar: eles dividem o problema em vários subproblemas semelhantes ao problema original, mas menores em tamanho, resolvem os subproblemas recursivamente e depois combinam essas soluções para criar uma solução para o problema original (CORMEN et al., 2001). O paradigma de dividir e conquistar envolve três etapas em cada nível da recursão: (i) dividir o problema em vários subproblemas; (ii) conquistar os subproblemas, resolvendo-os recursivamente – se os tamanhos dos subproblemas forem pequenos o suficiente, apenas resolva os subproblemas de maneira direta; (iii) combine as soluções dos subproblemas na solução do problema original.

Etapa de divisão:

- Com base na lista original, encontre o meio e separe-a em duas listas: esquerda_1 e direita_2.
- Com base na sublista esquerda_1, se a quantidade de elementos for maior que 1, encontre o meio e separe-a em duas listas: esquerda_1_1 e direta_1_1.
- Com base na sublista esquerda_1_1, se a quantidade de elementos for maior que 1, encontre o meio e separe-a em duas listas: esquerda_1_2 e direta_1_2.
- Repita o processo até encontrar uma lista com tamanho 1.
- Chame a etapa de merge.
- Repita o processo para todas as sublistas.

Para entender a etapa de junção do merge sort, suponha que você está vendo duas fileiras de crianças em ordem de tamanho. Ao olhar para a primeira criança de cada fila, é possível identificar qual é a menor e, então, colocá-la na posição correta. Fazendo isso sucessivamente, teremos uma fila ordenada com base nas duas anteriores.

Etapa de merge:

- Dadas duas listas, cada uma das quais contém 1 valor – para ordenar, basta comparar esses valores e fazer a troca, gerando uma sublista com dois valores ordenados.
- Dadas duas listas, cada uma das quais contém 2 valores – para ordenar, basta ir escolhendo o menor valor em cada uma e gerar uma sublista com quatro valores

ordenados.

- Repita o processo de comparação e junção dos valores até chegar à lista original, agora ordenada.

In [ ]:
```
def executar_merge_sort(lista):
    if len(lista) <= 1: return lista
    else:
        meio = len(lista) // 2
        esquerda = executar_merge_sort(lista[:meio])
        direita = executar_merge_sort(lista[meio:])
        return executar_merge(esquerda, direita)


def executar_merge(esquerda, direita):
    sub_lista_ordenada = []
    topo_esquerda, topo_direita = 0, 0
    while topo_esquerda < len(esquerda) and topo_direita < len(direita):
        if esquerda[topo_esquerda] <= direita[topo_direita]:
            sub_lista_ordenada.append(esquerda[topo_esquerda])
            topo_esquerda += 1
        else:
            sub_lista_ordenada.append(direita[topo_direita])
            topo_direita += 1
    sub_lista_ordenada += esquerda[topo_esquerda:]
    sub_lista_ordenada += direita[topo_direita:]
    return sub_lista_ordenada


lista = [10, 9, 5, 8, 11, -1, 3]
executar_merge_sort(lista)
```

Out[ ]:
```
[-1, 3, 5, 8, 9, 10, 11]
```

QUICKSORT (ORDENAÇÃO RÁPIDA)

Dado um valor em uma lista ordenada, à direita desse número existem somente números maiores que ele; e à esquerda, somente os menores. Esse valor, chamado de pivô, é a estratégia central no algoritmo quicksort. O algoritmo quicksort também trabalha com a estratégia de dividir para conquistar, pois, a partir do pivô, quebrará uma lista em sublistas (direita e esquerda) – a cada escolha do pivô e a cada quebra da lista, o processo de ordenação vai acontecendo. Para entendermos como o algoritmo funciona, suponha uma fila de pessoas que, por algum motivo, precisam ser ordenadas por ordem de tamanho. Pede-se para uma pessoa da fila se voluntariar para ter seu tamanho comparado (esse é o pivô). Com base nesse voluntário, todos que são menores que ele devem se dirigir para a esquerda e todos que são maiores para a direta. O voluntário está na posição ordenada. Vamos repetir o mesmo procedimento para os menores e maiores. A cada passo estamos dobrando o número de pessoas na posição final. A lógica é a seguinte:

- Primeira iteração: a lista original será quebrada através de um valor chamado de pivô. Após a quebra, os valores que são menores que o pivô devem ficar à sua esquerda e os maiores à sua direita. O pivô é inserido no local adequado, trocando a posição com o valor atual.

- Segunda iteração: agora há duas listas, a da direita e a da esquerda do pivô. Novamente são escolhidos dois novos pivôs e é feito o mesmo processo, de colocar à direita os menores e à esquerda os maiores. Ao final os novos pivôs ocupam suas posições corretas.
- Terceira iteração: olhando para as duas novas sublistas (direita e esquerda), repete-se o processo de escolha dos pivôs e separação.

Na última iteração, a lista estará ordenada, como resultado dos passos anteriores.

Dada uma lista a ser ordenada [10, 9, 5, 8], na primeira iteração o pivô é escolhido, o qual, no nosso caso, é o valor 8. Agora, para cada valor da lista, o pivô será comparado e acontecerá uma separação da lista em duas sublistas: à esquerda do pivô, os valores menores; e à direita, os valores maiores. No passo 1, movemos a linha que representa a sublista da direita, pois 10 é maior que o pivô. No passo 2, também movemos essa linha, pois 9 é maior que o pivô. No passo 3, movemos a linha da direita (porque sempre vamos fazer a comparação com os elementos da frente), mas movemos também a linha da esquerda, visto que 5 é menor que o pivô. Veja que, nesse movimento, os valores das posições são trocados: 5 passa a ocupar a posição da linha da esquerda, e o valor que ali estava vai para a posição da direita. Como não há mais elementos para comparar, uma vez que alcançamos o pivô, então ele é colocado no seu devido lugar, que está representado pela linha da esquerda. Para essa inserção, é feita a troca do valor que ali está com o valor do pivô. Veja que 8 passa a ocupar a posição 1, ao passo que 9 vai para a posição do pivô. Agora, todo o processo precisa se repetir, considerando-se, agora, as sublistas da esquerda e da direita. Na esquerda, temos uma sublista de tamanho 1 (valor 5), razão pela qual não há nada a ser feito. Por sua vez, na direita, temos a sublista [10, 9], razão pela qual se adota 9 como o pivô dela. Como o pivô é menor que o primeiro elemento dessa sublista, então o marcado da direita avança, mas o da esquerda também, fazendo a troca dos valores. Como não há mais comparações a serem feitas, o algoritmo encerra com a lista ordenada.

In [ ]:
```
def executar_quicksort(lista, inicio, fim):
    if inicio < fim:
        pivo = executar_particao(lista, inicio, fim)
        executar_quicksort(lista, inicio, pivo-1)
        executar_quicksort(lista, pivo+1, fim)
    return lista


def executar_particao(lista, inicio, fim):
    pivo = lista[fim]
    esquerda = inicio
    for direita in range(inicio, fim):
        if lista[direita] <= pivo:
            lista[direita], lista[esquerda] = lista[esquerda], lista[direita]
            esquerda += 1
    lista[esquerda], lista[fim] = lista[fim], lista[esquerda]
    return esquerda

lista = [10, 9, 5, 8, 11, -1, 3]
executar_quicksort(lista, inicio=0, fim=len(lista)-1)
```

Out[ ]: `[-1, 3, 5, 8, 9, 10, 11]`

Desafio

Como desenvolvedor em uma empresa de consultoria, você continua alocado para atender um cliente que precisa fazer a ingestão de dados de uma nova fonte e tratá-las. Você já fez uma entrega na qual implementou uma solução que faz o dedup em uma lista de CPFs, retorna somente a parte numérica do CPF e verifica se eles possuem 11 dígitos.

Os clientes aprovaram a solução, mas solicitaram que a lista de CPFs válidos fosse entregue em ordem crescente para facilitar o cadastro. Eles enfatizaram a necessidade de ter uma solução capaz de fazer o trabalho para grandes volumes de dados, no melhor tempo possível. Uma vez que a lista de CPFs pode crescer exponencialmente, escolher os algoritmos mais adequados é importante nesse caso.

Portanto, nesta nova etapa, você deve tanto fazer as transformações de limpeza e validação nos CPFs (remover ponto e traço, verificar se tem 11 dígitos e não deixar valores duplicados) quanto fazer a entrega em ordem crescente. Quais algoritmos você vai escolher para implementar a solução? Você deve apresentar evidências de que fez a escolha certa!

In [ ]:
```
criar_lista_dedup(lista=lista)

def merge_sort(lista):
    if len(lista) <= 1: return lista
    else:
        meio = len(lista) // 2
        esquerda = merge_sort(lista[:meio])
        direita = merge_sort(lista[meio:])
        return merge_sort(esquerda, direita)

def merge(esquerda, direita):
    sub_lista_ordenada = []
    topo_esquerda, topo_direita = 0, 0
    while topo_esquerda < len(esquerda) and topo_direita < len(direita):
        if esquerda[topo_direita] <= direita[topo_direita]:
            sub_lista_ordenada.append(esquerda[topo_esquerda])
            topo_esquerda += 1
        else:
            sub_lista_ordenada.append(direita[topo_direita])
            topo_direita += 1
    sub_lista_ordenada += esquerda[topo_esquerda:]
    sub_lista_ordenada += direita[topo_direita:]
    return sub_lista_ordenada

merge_sort(lista_cpfs)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 1
----> 1 criar_lista_dedup(lista=lista)
      3 def merge_sort(lista):
      4     if len(lista) <= 1: return lista

NameError: name 'criar_lista_dedup' is not defined
```